2011

Generative Type Abstraction and Type-level Computation

Stephanie Weirich
University of Pennsylvania, sweirich@cis.upenn.edu

Dimitrios Vytiniotis
Microsoft Research

Simon Peyton Jones
Microsoft Research

Stephan A. Zdancewic
University of Pennsylvania, stevez@cis.upenn.edu

Stephanie Weirich, Dimitrios Vytiniotis, Simon Peyton Jones, and Steve Zdancewic. **Generative Type Abstraction and Type-level Computation**. In *Proc. of the ACM Symposium on Principles of Programming Languages (POPL)*, 2011.
doi>10.1145/1925844.1926411

Generative Type Abstraction and Type-level Computation

Abstract
Modular languages support generative type abstraction, ensuring that an abstract type is distinct from its representation, except inside the implementation where the two are synonymous. We show that this well-established feature is in tension with the non-parametric features of newer type systems, such as indexed type families and GADTs. In this paper we solve the problem by using kinds to distinguish between parametric and non-parametric contexts. The result is directly applicable to Haskell, which is rapidly developing support for type-level computation, but the same issues should arise whenever generativity and non-parametric features are combined.

Disciplines
Computer Sciences

Comments
Stephanie Weirich, Dimitrios Vytiniotis, Simon Peyton Jones, and Steve Zdancewic. **Generative Type Abstraction and Type-level Computation**. In *Proc. of the ACM Symposium on Principles of Programming Languages (POPL)*, 2011.

doi>10.1145/1925844.1926411

Generative Type Abstraction and Type-level Computation

Stephanie Weirich
University of Pennsylvania
Philadelphia, PA, USA
sweirich@cis.upenn.edu

Dimitrios Vytiniotis
Simon Peyton Jones
Microsoft Research
Cambridge, UK
{dimitris,simonpj}@microsoft.com

Steve Zdancewic
University of Pennsylvania
Philadelphia, PA, USA
stevez@cis.upenn.edu

Abstract

Modular languages support *generative type abstraction*, ensuring that an abstract type is distinct from its representation, except inside the implementation where the two are synonymous. We show that this well-established feature is in tension with the *non-parametric features* of newer type systems, such as indexed type families and GADTs. In this paper we solve the problem by using kinds to distinguish between parametric and non-parametric contexts. The result is directly applicable to Haskell, which is rapidly developing support for type-level computation, but the same issues should arise whenever generativity and non-parametric features are combined.

Categories and Subject Descriptors D.3.3 [*Language Constructs and Features*]: Abstract data types; F.3.3 [*Studies of Program Constructs*]: Type structure

General Terms Design,Languages

Keywords Haskell, Newtype deriving, Type functions

1. Introduction

Generative type abstraction allows programmers to introduce new type constants in their programs that are *isomorphic* to existing types; examples include ML's module system [Milner et al. 1997; Pierce 2005, Ch. 8], and Haskell's **`newtype`** construct [Peyton Jones et al. 2003]. Type generativity is important because it supports modularity by enforcing *abstraction*: the *implementor* of a module can move freely between the abstract and representation types, whereas to the *client* of the module the two types are completely distinct.

There is growing interest in languages that support some form of *type-level computation* including Haskell's type classes [Hall et al. 1996] and indexed type families [Kiselyov et al. 2010]. However, there is a fundamental tension between type-level computation and generative type abstraction, at least in the latter's more flexible forms. To summarize very briefly, the conflict is this:

- To maximize re-use and convenience, it is very desirable for the implementor to be able to treat the abstract type A and its concrete representation type C as synonymous – we call this *flexible type generativity*.
- However, given type-level function F the result of $(F\ A)$ and $(F\ C)$ may differ, so A and C cannot be synonymous.

Resolving this conflict is the subject of this paper. Specifically our contributions are:

- We show in Section 2 that the naive combination of type generativity and non-parametric type-level features can violate soundness; a problem that already manifests in the Glasgow Haskell Compiler, and affects not only type-level non-parametric functions, but also other forms of non-parametric constructs, such as *generalized algebraic datatypes* (GADTs) [Cheney and Hinze 2003; Hinze et al. 2002; Peyton Jones et al. 2006; Xi et al. 2003].
- We formalize a solution to this problem that reconciles flexible type generativity and non-parametric type functions in Section 3. Our language, FC_2, builds on GHC's existing core language, System FC [Sulzmann et al. 2007], which supports erasable type-level coercions. The key ingredient in our solution is to employ kinds decorated with *roles* to mark the distinctions that type contexts may make.
- We prove that FC_2 programs are type safe, provided that user axioms and definitions give rise to consistent axiom sets (Section 4).
- We give sufficient conditions for showing the consistency of an axiom set. For our proofs, we introduce a rewrite system for types that is novel in two dimensions: rewriting (i) is *role-sensitive*, and (ii) need not be strongly normalizing (Section 4.2).
- We present a Haskell-specific result: we show how Haskell source programs that may involve non-parametric type contexts and flexible type generativity can be translated to yield provably consistent FC_2 axiom sets. (Section 5)
- Our core language is an *improvement* of System FC since it permits safe flexible type generativity, but also *unsaturated* type functions. Perhaps surprisingly, our language is additionally a significant *simplification* of the original System FC, which *removes* several of the original coercion constructs that we have identified to be encodable (even in the original System FC). We discuss these differences in Section 6.2.

For the sake of concreteness we build our presentation around Haskell and System FC, since this setting allows us to demonstrate our points with real code, instead of using a hypothetical λ-calculus. However, we stress that our work is applicable whenever flexible type generativity and non-parametric type-level features are combined. For example, the very same issues could arise in extensions of the ML module system. We discuss related work in Section 7.

POPL'11, January 26–28, 2011, Austin, Texas, USA.

Some of the technical material has been elided from this version of the paper for space. Although this paper is self contained, more details are available at `http://www.cis.upenn.edu/~sweirich/newtypes.pdf`.

2. The problems with generative type abstraction

Generative type abstraction is an extremely useful mechanism for enforcing abstraction barriers and for refining interfaces. To see this benefit, let us consider the Haskell incarnation of type generativity, namely **newtype** definitions. In Haskell, the programmer may declare a **newtype** `Age`, with concrete representation type `Int`, thus:

```
newtype Age = MkAge Int
```

The implementor uses the "data constructor" `MkAge` to coerce an `Int` to an `Age`, and pattern matching to effect the inverse coercion. For example:

```
addAge :: Age → Int → Age
addAge (MkAge a) n = MkAge (a+n)
```

The client, in contrast, can be prevented from making such conversions, by using the module system to hide the `MkAge` constructor:

```
module AgeModule( Age, addAge, ... ) where
  -- Age definition and implementation
```

2.1 Coercion lifting

In describing `MkAge`, we wrote "data constructor" in quotes because although it behaves in many ways like a data constructor, its cost model is different. Specifically, a **newtype** definition guarantees that the abstract type really is represented by the concrete type, so the runtime conversion cost is zero. That would not be true if `Age` were instead declared with **data** instead of **newtype**.

So `Age` can be coerced to an `Int`, and vice versa, for free—i.e. without runtime cost—because it *is* an `Int`. Notationally, we say that `Int` $\sim$ `Age`, where we use $\sim$ for type equality[1]. But what about, say, `Maybe Age`?

```
data Maybe a = Nothing | Just a
```

Obviously, `Maybe Age` should be freely coercible to `Maybe Int`, because the two are represented identically. Alas, in Haskell 98 one would have to write

```
cvtAge :: Maybe Age → Maybe Int
cvtAge t = mapMaybe (λ(MkAge a) → a) t
-- mapMaybe :: (a → b) → Maybe a → Maybe b
```

This is unsavory for several reasons: (a) it is tedious for the programmer; (b) it may be difficult to implement the necessary "map" function; (c) it is hard for the compiler to eliminate a runtime call to the map function (let alone to guarantee to do so) especially if the map function is recursive. As an example of (b) consider the mapping function for the type `T` shown below, with co- and contravariance, and higher kinds:

```
data T a f = T1 a
           | T2 (a → Int)
           | T3 (f (T a f) (T a f))
```

These difficulties are frustrating, because we know that, say, `T Age Maybe` is represented identically to `T Int Maybe`. So, let us imagine a hypothetical extension of Haskell that provides lifted coercions; that is, it implements the following rule:

> *Coercion lifting:* if for two types φ and ψ we have $\varphi \sim \psi$ (for example, if they are the abstract and concrete types of a **newtype** declaration), then $T\,\varphi \sim T\,\psi$ for any type constructor T.

In fact this extension is not so hypothetical, because such a rule is the basis of the so-called "newtype deriving" feature implemented by GHC. It is not difficult to use newtype deriving to define an identity function whose type is

```
cvt :: ∀{c}. c Int → c Age
```

ML supports coercion lifting even more directly: within a structure the abstract and the representation types are considered entirely interchangeable. For example in Standard ML one might say

```
signature AGE = sig
  type age
  val addAge : age →int →age
  ...
end

structure AgeModule :> AGE = struct
  type age = int
  fun addAge a n = a+n
  ...
end
```

Inside `AgeModule` the two types are synonymous, and so `addAge` need not convert in either direction.

However, the point of this paper is that the innocuous and obvious-seeming "lifting" of type identities becomes unsound when combined with type-level computation, as we show in the next section.

2.2 Type-level computation

One very popular extension to Haskell is that of Generalized Algebraic Data Types (GADTs) [Peyton Jones et al. 2006], with which we assume the reader is somewhat familiar. In GHC one could declare a GADT with two nullary constructors like this:

```
data K a where
  KAge :: K Age
  KInt :: K Int
```

Now, consider these definitions (using `cvt` from Section 2.1):

```
kint :: K Age
kint = cvt KInt

get :: K Age → String
get KAge = "Age"
```

Since `get`'s type signature declares that its argument is of type `K Age`, the patterns in `get` are exhaustive. But consider the call `(get kint)`. It is patently well typed, yet the pattern match in `get` will fail (recall that `cvt` is operationally the identity function), and if the compiler assumed otherwise a runtime crash could result.

In the last few years we have gone beyond GADTs, by extending GHC with type-level functions [Chakravarty et al. 2005a,b; Kiselyov et al. 2010]. The reader is urged to consult these papers for motivated examples of type functions, but for the purposes of this paper we content ourselves with a small but contrived example:

```
type family F a :: ⋆
type instance F Age = Char
type instance F Int = Bool
```

Here the type function `F` maps the type `Age` to the type `Char`, but it maps `Int` to `Bool`.

However, the existence of such a type-level function threatens not just pattern exhaustiveness but type soundness itself. Consider the type `Bool`. This type is equivalent to `F Int` by the equation

[1] There are too many sorts of "="!

for F; and by coercion lifting that should be equivalent to F Age; and that is equivalent to Char by the other equation for F. Altogether we can coerce Bool to Char, which is obvious nonsense.

What went wrong? Maybe it should be illegal for a type function to behave differently on two coercible types, such as Age and Int? But in fact Haskell programmers often use **newtype**s precisely so that they can give a different type-class instance (for comparison, say) for Age than for the underlying Int. Type functions are no different; indeed, they are often introduced as an "associated type" of a type class [Kiselyov et al. 2010], and hence, just as the type class distinguishes between the abstract and concrete type, so must the type function.

How else might we fix the problem? Perhaps, in the definition of coercion lifting we should not allow T to range over type functions such as F or GADTs such as K? Indeed we should not, but that is not enough. Consider

```
data TF a = MkTF (F a)
```

Now, should coercion lifting allow us to coerce TF Age to TF Int? Obviously not! Otherwise we could write

```
to :: Bool → TF Int      from :: TF Age → Char
to b = MkTF b            from (MkTF c) = c
```

and now the composition from ∘ cvt ∘ to is well-typed (via coercion lifting) but obviously unsound. So in the the definition of coercion lifting, as well as type functions and GADTs we must exclude data types like TF that use their type arguments non-parametrically.

2.3 Summary

At this point it should be clear that a naive combination of:

- type-level dispatch, whether by GADTs or by type functions
- unrestricted coercion lifting

simply does not work. This interaction was far from obvious to us initially, and indeed GHC has a well-documented type-soundness bug[2] that arises *directly* from this unforeseen interaction. Yet both type-level dispatch and coercion lifting (suitably restricted) are valuable. The purpose of this paper is to show how they may be soundly combined.

This problem is important not only because it arises in GHC (which is our main source of motivation), but also because the same issues will arise in *any* type system that combines type-level dispatch and coercion lifting. Haskell is the first programming language that has pushed the type system far enough for these problems to arise in practice, but others may do so in the future. However, these different languages may expose the coercions between abstract and concrete types in different ways, possibly explicitly (as in Haskell) or implicitly (as in ML), in a way that is intimately connected with type inference. To avoid the complications associated with type inference, in this paper, we focus on an *intermediate language*, in which (runtime-erasable) coercions are explicit. Whether they come directly from the source program, or from elaboration by the type checker, is secondary.

3. The FC_2 language: codes versus types

Our goal is to resolve the tension between *generative types*, which allow programmers to express the intent that two types have identical representations, and *type functions*, which can distinguish two types even if they have the same underlying representation.

The inspiration for our solution comes from encodings of generic programming in dependently typed languages [Benke et al. 2003; Dybjer 2000]. These encodings use "codes" to represent types as a form of data. For example, the code IntCode may be the code for the type Int. Nonparametric functions branch on codes and thus have a different type from parametric functions. In this context, we can view Age and IntCode as two different codes that both map to the type Int. However, this encoding requires significant syntactic overhead as we must have a code for every type and must explicitly map codes to types when they are used to classify expressions.

However, the important distinction between codes and types is that they have *different definitions of equality*. In the encoding above, the codes Age and IntCode are different codes, but their interpretations are equal types. Therefore, we use this idea in FC_2 to define *roles*, which support different notions of equality for the same data. In this language, Age and Int are distinct when viewed at role *code* but equal when viewed at role *type*. Code equality is used to reason about the meaning of type-indexed functions and is finer-grained than type equality, which is used to determine which type coercions are safe. Importantly, FC_2 distinguishes type functions by what equivalences they respect. Parametric functions respect the role *type*, whereas functions that distinguish between Age and Int do not.

3.1 FC_2: an overview

These ideas are best explained in terms of an intermediate language that exposes the differences between the code and type roles and makes explicit the uses of the two kinds of equality mentioned above. Thus, the remainder of this section describes FC_2, our new variant of System FC [Sulzmann et al. 2007]—a model of the intermediate language used in GHC. As such, it is expressive enough to capture indexed type functions, **newtype** and **newtype deriving**, GADTs, existential and nested datatypes, and much more.

Figure 1 summarizes the syntax of FC_2, which, at the term level (e), is just the polymorphic lambda calculus with two extensions. First, FC_2 provides polymorphic datatypes, introduced by data constructors K. These datatypes are eliminated using a **case** construct that should be familiar from Haskell or ML-style functional programming—we describe how datatypes and **case** are typechecked in Section 3.5.

Second, FC_2 includes first-class proofs of type equality that witness safe coercions introduced during compilation. Programs in FC_2 can abstract over coercions reflecting a particular type equality (written $\Lambda c : \varphi_1 \sim \varphi_2.e$), pass a coercion as an argument to such a function (written $e\,\gamma$), and use a coercion to cast a term from one type to another (written $e \triangleright \gamma$). These explicit coercions, written γ, make typechecking FC_2 programs *syntax-directed*—that is, the syntax of an FC_2 term encodes its typing derivation. Why is this important? The idea is that the compiler's front end performs perhaps-complex type inference on the source program, and records the proofs generated by inference directly in the syntax of the FC_2 intermediate language. The optimiser transforms FC_2 terms, perhaps radically. At any point one can check the consistency of the resulting FC_2 program using a simple, fast, syntax-directed typechecker; this consistency check has proven to be an extremely powerful aid to getting the compiler right. It is just as easy to find the type of an arbitrary FC_2 term, an ability that is used extensively inside GHC.

Figure 2 shows the typing rules for the terms of FC_2. The first five rules, EEVAR through ETAPP, are completely standard. We defer explanation of the remaining rules until we build up technical machinery having to do with FC_2's kind-level distinction between codes and types and the rules by which explicit coercions can themselves be combined. We describe these aspects of FC_2 next.

[2] http://hackage.haskell.org/trac/ghc/ticket/1496

η	::=	$\star \mid \kappa \to \eta$	kind
R	::=	$\mathsf{C} \mid \mathsf{T}$	role
κ	::=	η / R	kind and role
H	::=		type constants
	$\mid$	T	datatypes
	$\mid$	F	functions/newtypes
	$\mid$	$(\to)$	arrow
	$\mid$	$(\sim_\eta)$	equality
φ, σ, ψ	::=		codes and types
	$\mid$	a	variables
	$\mid$	H	constants
	$\mid$	$\varphi_1\, \varphi_2$	application
	$\mid$	$\forall a{:}\kappa.\varphi$	polymorphism
γ	::=		coercion proof
	$\mid$	$c\, \overline{\varphi}$	assumption
	$\mid$	φ	reflexivity
	$\mid$	$\mathbf{sym}\, \gamma$	symmetry
	$\mid$	$\gamma_1\, ;\, \gamma_2$	transitivity
	$\mid$	$\gamma_1\, \gamma_2$	application
	$\mid$	$\mathbf{nth}\, k\, \gamma$	injectivity
	$\mid$	$\forall a{:}\kappa.\gamma_2$	polymorphism
	$\mid$	$\gamma @ \varphi$	instantiation
e	::=		expressions
	$\mid$	x	variable
	$\mid$	$\lambda x{:}\sigma.e$	abstraction
	$\mid$	$e_1\, e_2$	application
	$\mid$	$\Lambda a{:}\kappa.e$	type abstraction
	$\mid$	$e\, \varphi$	type application
	$\mid$	K	data constructor
	$\mid$	$\mathbf{case}_\sigma\ e\ \mathbf{of}\ brs$	case analysis
	$\mid$	$\Lambda c{:}\varphi_1 \sim \varphi_2.e$	proof abstraction
	$\mid$	$e\, \gamma$	proof application
	$\mid$	$e \triangleright \gamma$	coercion
brs	::=	$\overline{K_i \Rightarrow e_i}^{\,i \in 1..n}$	branches
bnd	::=		binding
	$\mid$	$a{:}\kappa$	type variable
	$\mid$	$H{:}\eta$	type constant
	$\mid$	$c{:}\Delta.\, \varphi_1 \sim \varphi_2 / R$	coercion
	$\mid$	$x{:}\sigma$	term variable
	$\mid$	$K{:}\sigma$	data constructor
Γ	::=	$\cdot \mid \Gamma, bnd$	context
Δ	::=	$\cdot \mid a{:}\kappa, \Delta$	type context
ρ	::=	$e \mid \varphi \mid \gamma$	datacon argument
Θ	::=		telescopes
	$\mid$	$\cdot$	empty
	$\mid$	σ, Θ	expression type
	$\mid$	$a{:}\kappa, \Theta$	type variable
	$\mid$	$\varphi_1 \sim \varphi_2, \Theta$	equality
v	::=	$\lambda x{:}\sigma.e \mid \Lambda a{:}\kappa.e$	values
	$\mid$	$\Lambda c{:}\varphi_1 \sim \varphi_2.e$	
	$\mid$	$K\, \overline{\varphi}\, \overline{\rho}$	
τ	::=	$T \mid (\to) \mid (\sim_\eta)$	value types
	$\mid$	$\forall a{:}\kappa.\varphi \mid \tau\, \varphi$	

Figure 1. Syntax

$\boxed{\Gamma \vdash e : \sigma}$

$$\frac{x{:}\sigma \in \Gamma \quad \vdash \Gamma}{\Gamma \vdash x : \sigma}\ \textsc{EEVar}$$

$$\frac{\Gamma, x{:}\sigma_1 \vdash e : \sigma_2}{\Gamma \vdash \lambda x{:}\sigma_1.e : \sigma_1 \to \sigma_2}\ \textsc{EEAbs}$$

$$\frac{\Gamma \vdash e_1 : \sigma_1 \to \sigma_2 \quad \Gamma \vdash e_2 : \sigma_1}{\Gamma \vdash e_1\, e_2 : \sigma_2}\ \textsc{EEApp}$$

$$\frac{\Gamma, a{:}\kappa \vdash e : \sigma}{\Gamma \vdash \Lambda a{:}\kappa.e : \forall a{:}\kappa.\sigma}\ \textsc{ETAbs}$$

$$\frac{\Gamma \vdash e : \forall a{:}\kappa.\sigma \quad \Gamma \vdash \varphi : \kappa}{\Gamma \vdash e\, \varphi : \sigma[a \mapsto \varphi]}\ \textsc{ETApp}$$

$$\frac{\begin{array}{l} T{:}\overline{\kappa} \to \star \in \Gamma \\ K{:}\forall\, \overline{a}{:}\overline{\kappa}.\forall\, \Theta.T\, \overline{a} \in \Gamma \quad \vdash \Gamma \end{array}}{\Gamma \vdash K : \forall\, \overline{a}{:}\overline{\kappa}.\forall\, \Theta.T\, \overline{a}}\ \textsc{EDataCon}$$

$$\frac{\begin{array}{l} \Gamma \vdash e : T\, \overline{\varphi} \quad \Gamma \vdash \sigma : \star/\mathsf{T} \\ \text{for each } K_i \in \mathrm{Constr}_\Gamma(T) \\ \quad K_i{:}\forall\, \overline{a}{:}\overline{\kappa}.\psi_i \in \Gamma \\ \quad \psi_i[\overline{a} \mapsto \overline{\varphi}] = \forall\, \Theta_i.T\, \overline{\varphi} \\ \quad \Gamma \vdash e_i : \forall\, \Theta_i.\sigma \end{array}}{\Gamma \vdash \mathbf{case}_\sigma\ e\ \mathbf{of}\ \overline{K_i \Rightarrow e_i}^{\,i} : \sigma}\ \textsc{ECase}$$

$$\frac{\Gamma, c{:}\varphi_1 \sim \varphi_2 / \mathsf{C} \vdash e : \sigma}{\Gamma \vdash \Lambda c{:}\varphi_1 \sim \varphi_2.e : (\varphi_1 \sim \varphi_2) \Rightarrow \sigma}\ \textsc{ECAbs}$$

$$\frac{\begin{array}{l} \Gamma \vdash e : (\varphi_1 \sim \varphi_2) \Rightarrow \sigma \\ \Gamma \vdash \gamma : \varphi_1 \sim \varphi_2 \in \eta/\mathsf{C} \end{array}}{\Gamma \vdash e\, \gamma : \sigma}\ \textsc{ECApp}$$

$$\frac{\Gamma \vdash e : \sigma_1 \quad \Gamma \vdash \gamma : \sigma_1 \sim \sigma_2 \in \star/\mathsf{T}}{\Gamma \vdash e \triangleright \gamma : \sigma_2}\ \textsc{ECoerce}$$

Figure 2. Typing rules

$\boxed{R_1 \preceq R_2}$ $\quad \dfrac{}{\mathsf{C} \preceq \mathsf{T}}\ \textsc{RSub} \qquad \dfrac{}{R \preceq R}\ \textsc{RRefl}$

$\boxed{\Gamma \vdash \varphi : \kappa}$

$$\frac{a{:}\eta/R_1 \in \Gamma \quad R_1 \preceq R_2 \quad \vdash \Gamma}{\Gamma \vdash a : \eta/R_2}\ \textsc{PVar}$$

$$\frac{H{:}\eta \in \Gamma \quad \vdash \Gamma}{\Gamma \vdash H : \eta/R}\ \textsc{PConst}$$

$$\frac{\begin{array}{l} \Gamma \vdash \varphi_1 : (\eta_1/R_2 \to \eta_2)/R_1 \\ \Gamma \vdash \varphi_2 : \eta_1/min(R_1, R_2) \end{array}}{\Gamma \vdash \varphi_1\, \varphi_2 : \eta_2/R_1}\ \textsc{PApp}$$

$$\frac{\Gamma, a{:}\kappa \vdash \varphi : \star/R}{\Gamma \vdash \forall a{:}\kappa.\varphi : \star/R}\ \textsc{PAll}$$

$\boxed{\Gamma \vdash \overline{\varphi} : \overline{\kappa}}$

$$\frac{}{\Gamma \vdash \cdot : \cdot}\ \textsc{PNil} \qquad \frac{\Gamma \vdash \varphi : \kappa \quad \Gamma \vdash \overline{\varphi} : \overline{\kappa}}{\Gamma \vdash \varphi, \overline{\varphi} : \kappa, \overline{\kappa}}\ \textsc{PCons}$$

Figure 3. Kinding

3.2 FC_2 types and kinds

Types in FC_2 are classified by pairs κ of the form η/R, where the *kind* η ensures (as usual) that types are well-formed structurally, and the *role* R that determines the precision at which they can be analyzed. Codes (which distinguish `Age` and `Int`) have role C, whereas types (which identify them) have role T. This syntax is summarized at the top of Figure 1, while the kinding rules are in Figure 3.

The distinction between codes and types allows us to give informative kinds to type constructors:

- The `Maybe` type (Section 2.1) has kind $\star/\mathsf{T} \to \star$, indicating that `Maybe` treats its argument *parametrically*.
- The types `K`, `F`, and `TF` (Section 2.2) all use *type indexing* and therefore have kind $\star/\mathsf{C} \to \star$.

These kinds in turn support the key insight of this paper: *it is only safe to lift coercions through functions with parametric kinds*. So `Maybe Age` $\sim$ `Maybe Int` holds but `TF Age` $\not\sim$ `TF Int`.

The rules for constructing coercions are given in Section 3.3, but first we must deal with the kinding rules for types. The critical rule is PAPP in Figure 3, which deals with type application. It is quite conventional except for the treatment of roles. Consider these type declarations:

```
data TF1 a = MkTF1 (F (Maybe a)
data TF2 a = MkTF2 (Maybe (F a))
data TF3 a = MkTF3 (Maybe (Maybe a))
```

In the first two cases, type variable `a` is used non-parametrically (intuitively, under a type function), so `TF1` $: \star/\mathsf{C} \to \star$, and similarly for `TF2`. In contrast, `TF3` treats `a` parametrically, so `TF1` $: \star/\mathsf{T} \to \star$. The unusual "$min$" in rule PAPP achieves these kindings by combining the role of the context and the argument role of the function to get the role of the argument.

In fact, min is just the least upper bound induced by an inclusion relation $\mathsf{C} \preceq \mathsf{T}$ on roles (see the top of Figure 3). This inclusion makes explicit the fact that code equality implies type equality but not vice-versa. For example, consider

```
data TF4 a = MkTF4 (F a) a
```

Here `a` is used both non-parametrically (as an argument of `F`) and parametrically (as a plain argument of `MkTF4`). So `a` has role $\star/\mathsf{C}$, which makes sense; just because a type *can* be analyzed does not mean that it *must* be.

Despite these non-standard kinds, the types of FC_2 are mostly standard. Codes φ and types σ are drawn from the same syntax (see Figure 1), although we use two different metavariables as a reminder of the intended role. The type language includes type variables a, type constants H, applications $\varphi_1\ \varphi_2$, and polymorphic types $\forall a{:}\kappa.\sigma$.

Type constants, H, include datatypes T, and type functions F. For the most part, datatypes and type functions are treated uniformly, but there are two important distinctions:

- Datatypes must be injective, while type functions need not be. Injectivity is important because equalities between applications of injective functions imply equalities between their arguments; see rule CNTHT in Section 3.3.
- Datatypes are inhabited by values, but type functions are not—there are no values v with types that are headed by F. (There are *coerced values* with such types, but no raw values.) This distinction is key to the definition of consistency in Section 4.1.

Newtypes are not inhabited by raw values, so we treat them like type functions, ranged over by F. Unlike type functions, however, newtypes *are* injective at role C—after all, the essence of generativity is that newtypes create a fresh constant—but for now we will not take advantage of that fact, leaving it for discussion in Section 6.1.

The set of type constants also includes the familiar arrow type constructor $(\to)$, and a kind-indexed family of constructors $(\sim_\eta)$, which construct functions that abstract over coercions. The kinds of these constants are:

$$\begin{array}{lcl}(\to) & : & \star/\mathsf{T} \to \star/\mathsf{T} \to \star \\ (\sim_\eta) & : & \eta/\mathsf{C} \to \eta/\mathsf{C} \to \star/\mathsf{T} \to \star\end{array}$$

(We discuss in Sections 3.3 and 3.6 why the kind of $(\sim_\eta)$ must require role C for its first two arguments.) Type constants are generally applied prefix, but for these two constants we define infix syntactic sugar:

$$\begin{array}{lcl}\sigma \to \sigma' & \equiv & (\to)\ \sigma\ \sigma' \\ (\varphi_1 \sim \varphi_2) \Rightarrow \sigma & \equiv & (\sim_\eta)\ \varphi_1\ \varphi_2\ \sigma\end{array}$$

In the latter case, because the syntactic sugar lacks the η annotation, we only use this notation in contexts where the kind of φ_1 and φ_2 is irrelevant. Rule ECABS follows this convention—it shows that this family of type constructors is used to give a type to terms of the form $\Lambda c : \varphi_1 \sim \varphi_2.e$, which abstracts over an equality proof in the body e. Note that this rule applies only to code equalities; abstraction over type equalities is not needed for compilation of Haskell and permitting it, while straightforward, would require extra syntactic complexity.

Figure 3 defines the kinding rules for FC_2 using judgments of the form $\Gamma \vdash \varphi : \kappa$. Here, the context Γ maps type variables to their kind/role pairs and type constants and type functions to their kinds. This judgment relies on a judgment $\vdash\ \Gamma$ (shown in the extended version) that checks that contexts are well formed. These checks ensure that constants $(\to)$ and $(\sim_\eta)$ have the right kinds, that data constructors and term variables have well-formed types and that coercion axioms introduce equalities between well-formed codes and types.

Rule PCONST allows a type constant to play either role. Although a type constant may be given role C, its *kind* may well involve arguments with role T; for example, see the signature for $(\to)$ above. It follows that closed types can play both roles, independently of whether they contain newtypes or type functions. For example `(Int,Age)` $: \star/\mathsf{C}$ and `(Int,Age)` $: \star/\mathsf{T}$ both hold.

Together these kinding rules implement a subsumption relation that includes codes into the language of types:

LEMMA 1. *If* $\Gamma \vdash \varphi : \eta/\mathsf{C}$ *then* $\Gamma \vdash \varphi : \eta/\mathsf{T}$.

On the other hand, types have only one kind regardless of their role:

LEMMA 2. *If* $\Gamma \vdash \varphi : \eta_1/R_1$ *and* $\Gamma \vdash \varphi : \eta_2/R_2$ *then* $\eta_1 = \eta_2$.

Note that the kinding judgment, like term typing, is syntax-directed (see the remarks in Section 3.1). In particular, the role component R of the κ in this judgment is treated as *input* to the algorithm, and the η is an *output*—the context in which φ is used determines what the role should be. The only interesting case from this perspective is PAPP, in which φ_1 must be checked first to obtain R_2 so that the minimum of R_1 and R_2 can be passed as an input when checking φ_2.

3.3 Coercions and equality

In FC_2 a *coercion* is a proof term that witnesses the equality of two types. Coercions are used to change the type of a term, thus (Figure 2):

$$\frac{\Gamma \vdash e : \sigma_1 \quad \Gamma \vdash \gamma : \sigma_1 \sim \sigma_2 \in \star/\mathsf{T}}{\Gamma \vdash e \triangleright \gamma : \sigma_2}\ \text{ECOERCE}$$

Here, γ is a *coercion* witnessing the equality $\sigma_1 \sim \sigma_2$ in role T; given that e has type σ_1, we can use γ to let us treat the term as hav-

$\boxed{\Gamma \vdash \gamma : \varphi_1 \sim \varphi_2 \in \kappa}$

$$\frac{(c : \overline{a} : \overline{\kappa}.\, \varphi_1 \sim \varphi_2 / R_1) \in \Gamma \quad \Gamma \vdash \varphi_1 : \eta / R_1 \quad \Gamma \vdash \overline{\psi} : \overline{\kappa} \quad R_1 \preceq R_2}{\Gamma \vdash c\,\overline{\psi} : \varphi_1[\overline{a} \mapsto \overline{\psi}] \sim \varphi_2[\overline{a} \mapsto \overline{\psi}] \in \eta / R_2} \text{ CASSM}$$

$$\frac{\Gamma \vdash \varphi : \kappa}{\Gamma \vdash \varphi : \varphi \sim \varphi \in \kappa} \text{ CREFL}$$

$$\frac{\Gamma \vdash \gamma : \varphi_1 \sim \varphi_2 \in \kappa}{\Gamma \vdash \mathbf{sym}\,\gamma : \varphi_2 \sim \varphi_1 \in \kappa} \text{ CSYM}$$

$$\frac{\Gamma \vdash \gamma_1 : \varphi_1 \sim \varphi_2 \in \kappa \quad \Gamma \vdash \gamma_2 : \varphi_2 \sim \varphi_3 \in \kappa}{\Gamma \vdash \gamma_1\,;\,\gamma_2 : \varphi_1 \sim \varphi_3 \in \kappa} \text{ CTRANS}$$

$$\frac{\Gamma \vdash \gamma_1 : \varphi_1 \sim \varphi_2 \in (\eta_1 / R_2 \to \eta_2) / R_1 \quad \Gamma \vdash \gamma_2 : \psi_1 \sim \psi_2 \in \eta_1 / min(R_1, R_2)}{\Gamma \vdash \gamma_1\ \gamma_2 : \varphi_1\,\psi_1 \sim \varphi_2\,\psi_2 \in \eta_2 / R_1} \text{ CAPP}$$

$$\frac{\Gamma \vdash \gamma : T\,\overline{\varphi} \sim T\,\overline{\psi} \in \eta / \mathsf{T} \quad T : \overline{\kappa} \to \star \in \Gamma \quad \eta' / R_1 = nth\ k\ \overline{\kappa} \quad R_1 \preceq R_2}{\Gamma \vdash \mathbf{nth}\,k\,\gamma : nth\ k\ \overline{\varphi} \sim nth\ k\ \overline{\psi} \in \eta' / R_2} \text{ CNTHT}$$

$$\frac{\Gamma, a : \kappa \vdash \gamma_2 : \varphi_1 \sim \varphi_2 \in \star / R}{\Gamma \vdash \forall a : \kappa.\gamma_2 : \forall a : \kappa.\varphi_1 \sim \forall a : \kappa.\varphi_2 \in \star / R} \text{ CALL}$$

$$\frac{\Gamma \vdash \gamma : \forall a : \kappa.\varphi_1 \sim \forall a : \kappa.\varphi_2 \in \star / R \quad \Gamma \vdash \psi : \kappa}{\Gamma \vdash \gamma @ \psi : (\varphi_1[a \mapsto \psi]) \sim (\varphi_2[a \mapsto \psi]) \in \star / R} \text{ CINST}$$

$\boxed{\Gamma \vdash \overline{\gamma} : \overline{\varphi}_1 \sim \overline{\varphi}_2 \in \overline{\kappa}}$

$$\frac{\vdash \Gamma}{\Gamma \vdash \cdot : \cdot \sim \cdot \in \cdot} \text{ CNIL}$$

$$\frac{\Gamma \vdash \gamma : \varphi_1 \sim \varphi_2 \in \kappa \quad \Gamma \vdash \overline{\gamma} : \overline{\varphi}_1 \sim \overline{\varphi}_2 \in \overline{\kappa}}{\Gamma \vdash \gamma, \overline{\gamma} : \varphi_1, \overline{\varphi}_1 \sim \varphi_2, \overline{\varphi}_2 \in \kappa, \overline{\kappa}} \text{ CCONS}$$

Figure 4. Coercions

ing type σ_2. At compile time, these explicit coercions ensure that typechecking FC_2 programs is completely syntax directed. Such coercions have no run-time effect: they will be erased by the compiler before the program is run. Nevertheless, FC_2's operational semantics includes coercions, thereby allowing us to establish type safety using standard techniques (Section 3.6).

The translation of a source program into FC_2 may extend the type environment Γ with new equality axioms. For example, the `Age` **`newtype`** definition generates the axiom (where `mkAge` is a coercion constant c):

$$\texttt{mkAge} : \texttt{Age} \sim \texttt{Int} / \mathsf{T}$$

Note that ECOERCE requires σ_1 and σ_2 to be equal when considered in role T, which is consistent with the idea that type equality determines when it is safe to coerce. On the other hand, source programs can also introduce code equalities. For example the type function `F` (Section 2.2) generates the two axioms:

$$\texttt{axF1} : \texttt{F Int} \sim \texttt{Bool} / \mathsf{C}$$
$$\texttt{axF2} : \texttt{F Age} \sim \texttt{Char} / \mathsf{C}$$

However, all code axioms can be used to prove type equalities, as code equality is a refinement of type equality.

More generally we permit axiom *schemes*. For example, the source language declaration

```
type instance F(Maybe a) = (a,a)
```

would create the axiom scheme

$$\texttt{axF3} : (a : \star / \mathsf{C}).\, \texttt{F (Maybe a)} \sim \texttt{(a,a)} / \mathsf{C}$$

In general, as shown in Figure 1, the context Γ includes bindings of the form $c : \Delta.\, \varphi_1 \sim \varphi_2 / R$ for coercion axioms. The metavariable Δ stands for a type variable context: a list of quantified type variable bindings of the form $a : \kappa$. The same binding form is used both for axioms introduced at top level, and (with empty Δ) for local assumptions introduced in ECABS (Figure 2).

Of course it is important to know that the top-level axioms are *consistent*—it would be unsound to assert that $\texttt{Bool} \sim \texttt{Char} / \mathsf{T}$, for example. Section 4 gives a sufficient set of conditions for ensuring that source programs generate consistent axioms.

Next, we need a way to compose coercions together to construct other coercions. Our goal is to have rules that allow the creation of composite coercions such as:

$$\texttt{List mkAge} \ : \ \texttt{List Age} \sim \texttt{List Int} / \mathsf{T}$$
$$\texttt{List axF2} \ : \ \texttt{List (F Int)} \sim \texttt{List Bool} / \mathsf{T}$$

On the other hand, the coercion formation rules should disallow the formation of a coercion of the form $\gamma_3 : \texttt{F Age} \sim \texttt{F Int} / \mathsf{T}$, which creates the unsoundness described in Section 2.2.

Figure 1 gives the syntax of coercion terms, γ, and Figure 4 gives their typing rules. Coercions are typechecked using the judgment: $\Gamma \vdash \gamma : \varphi_1 \sim \varphi_2 \in \eta / R$, which asserts that the type of a coercion γ is an equality proposition $\varphi_1 \sim \varphi_2 \in \eta / R$. This proposition in turn implies that φ_1 and φ_2 both have kind η and are equal in role R. It is technically convenient to include η in the syntax of the judgement to enforce that both types have the same kind. However, this component is not always relevant, so we sometimes omit the $\in \eta$ part, as we have done in the examples above.

Rule CASSM instantiates an axiom scheme with types $\overline{\psi}$, using an auxiliary judgment $\Gamma \vdash \overline{\psi} : \overline{\kappa}$ defined at the bottom of Figure 3 to ensure that each variable is instantiated with a type of the matching kind and role. The notation $\overline{a} : \overline{\kappa}$ zips together a list of type variables and a list of kinds to create a type variable context Δ. These two lists must have the same length for the notation to be well-defined. The notation $\varphi[\overline{a} \mapsto \overline{\psi}]$ applies a multi-substitution of the types $\overline{\psi}$ for each of the corresponding variables in the list $\overline{a}$.

Rule CREFL shows that any type φ can be lifted to a reflexive coercion φ, while CSYM and CTRANS add symmetry and transitivity, ensuring that equality is an equivalence relation. The rules CAPP and CALL extend equality compatibly over applications and polymorphic types; their structure is analogous to the corresponding kinding rules in Figure 3. Rule CAPP is particularly important, because it implements the key coercion lifting idea we discussed in Section 2.1, using kinds to prevent the formation of the bogus coercion

$$\texttt{FmkAge} : \texttt{F Age} \sim \texttt{F Int} / \mathsf{T}$$

To see why, recall that `F` has kind $\star / \mathsf{C} \to \star$, but the `mkAge` axiom holds only at role T—type equalities cannot be lifted through code functions. Another example of a coercion that is correctly rejected by the application rule (because of the kind of $(\sim_\eta)$) is

$$(\sim_\star)\ \texttt{mkAge Int}\ \sigma$$

This coercion proves $(\texttt{Age} \sim \texttt{Int}) \Rightarrow \sigma \sim (\texttt{Int} \sim \texttt{Int}) \Rightarrow \sigma$, an equality that could be used to introduce a bogus assumption that $\texttt{Age} \sim \texttt{Int} / \mathsf{C}$ and satisfy it with reflexivity for `Int`.

As well as composing coercions to witness the equality of bigger types, it is also essential to do the reverse: to decompose equalities over complex types to give equalities of simpler types.

Decomposition is required by FC_2's operational semantics (Section 3.6), and it also makes the language usefully more expressive. Rule CINST allows equalities between polymorphic types to be instantiated. The other, more important decomposition rule is CNTHT, which decomposes the application of a datatype constant to arguments. For example, given a coercion γ : `List Int` $\sim$ `List a`$/\mathsf{T}$ we can use $\mathbf{nth}\ 0\ \gamma$ to conclude that `Int` $\sim$ `a`$/\mathsf{T}$. The soundness of this rule depends on the fact that datatypes are *injective*. In general, type functions are not, and hence CNTHT is restricted to datatypes T.

In rule CNTHT, the notation $T\,\overline{\varphi}$ abbreviates the multi-application $((T\,\varphi_1)\ldots\varphi_n)$ for $\varphi_1\ldots\varphi_n$ in $\overline{\varphi}$. In the conclusion of the rule, the notation $nth\ k\ \overline{\psi}$, accesses the kth element of the sequence of types, and $nth\ k\ \overline{\kappa}$, accesses the kind of the k^{th} variable binding. Both of these notations are undefined if k is not less than the length of the sequence. The sequence of types $\overline{\kappa}$ is determined by the kind of the datatype, using the following notation.

DEFINITION 3. *We define $\overline{\kappa} \to \eta$ by induction on $\overline{\kappa}$ as follows:*

$$\begin{array}{lcl} \cdot \to \eta & = & \eta \\ \kappa, \overline{\kappa} \to \eta & = & \kappa \to (\overline{\kappa} \to \eta) \end{array}$$

The coercion judgment satisfies a number of sanity checking properties. First, coercions are between well-formed types.

LEMMA 4 (Coercion regularity). *If $\Gamma \vdash \gamma : \varphi_1 \sim \varphi_2 \in \kappa$ then $\Gamma \vdash \varphi_1 : \kappa$ and $\Gamma \vdash \varphi_2 : \kappa$.*

Next, each coercion proof is for one pair of codes/types.

LEMMA 5 (Unique types). *If $\Gamma \vdash \gamma : \varphi_1 \sim \varphi_2 \in \eta/R$ and $\Gamma \vdash \gamma : \varphi_1' \sim \varphi_2' \in \eta'/R$ then $\varphi_1 = \varphi_1'$ and $\varphi_2 = \varphi_2'$ and $\eta = \eta'$.*

Finally, equality for Cs is a refinement of that for Ts; that is, code equality implies type equality, but not vice versa.

LEMMA 6. *If $\Gamma \vdash \gamma : \varphi_1 \sim \varphi_2 \in \eta/\mathsf{C}$ then $\Gamma \vdash \gamma : \varphi_1 \sim \varphi_2 \in \eta/\mathsf{T}$.*

3.4 Coercion lifting

The application rule CAPP described in the previous section allows us to lift equalities through arbitrary type constructors; that is, for all datatypes T of kind $\star/\mathsf{T} \to \star$, we have a coercion that shows T `Age` $\sim$ T `Int` $\in \star/\mathsf{T}$.

In fact, this notion of coercion lifting is not restricted to datatypes (like `List`), but is available for more general contexts. More precisely, given an arbitrary type σ with free variable a of kind $\star/\mathsf{T}$, we can also create a coercion $\sigma[a \mapsto$ `Age`$] \sim \sigma[a \mapsto$ `Int`$] \in \star/\mathsf{T}$.

We create such coercions with the *lifting operation*. This operation replaces type variables by coercions in types to produce a new coercion, relying on the fact that the syntax of types is a subset of the syntax of coercions:

DEFINITION 7 (Lifting Operation). *Define the lifting operation, written $\varphi[a \mapsto \gamma]$, by induction on φ.*

$$\begin{array}{lcll} a[a \mapsto \gamma] & = & \gamma & \\ b[a \mapsto \gamma] & = & b & \text{when } a \neq b \\ H[a \mapsto \gamma] & = & H & \\ (\varphi\,\psi)[a \mapsto \gamma] & = & (\varphi[a \mapsto \gamma])\ (\psi[a \mapsto \gamma]) & \\ (\forall b{:}\kappa.\sigma)[a \mapsto \gamma] & = & \forall b{:}\kappa.(\sigma[a \mapsto \gamma]) & \end{array}$$

The lifting operation produces a valid result as long as the role of the lifted coercion matches the role of the type variable in the type.

LEMMA 8 (Lifting). *If $\Gamma, a{:}\eta/R \vdash \sigma : \kappa$ and $\Gamma \vdash \gamma : \varphi \sim \varphi' \in \eta/R$, then $\Gamma \vdash \sigma[a \mapsto \gamma] : \sigma[a \mapsto \varphi] \sim \sigma[a \mapsto \varphi'] \in \kappa$.*

In other words, if a were used in some indexed context in σ, that is, if its role were C, then we would not be able to lift the coercion `Age` $\sim$ `Int` $\in \star/\mathsf{T}$ in σ. We generalize lifting to replace multiple type variables simultaneously in the obvious way, with notation $\sigma[\overline{a} \mapsto \overline{\gamma}]$.

3.5 Pattern matching and datatypes

FC_2 includes a formalization of recursive datatypes. These datatypes include all GHC extensions to standard datatypes: empty datatypes, nested datatypes, existential types, first-class polymorphism and GADTs. Both datatypes T and data constructors K must be declared in a context Γ before they can be used. For example, using the syntax for Γ in Figure 1, the declarations for the data constructors of `List` are:

$$\begin{array}{lcl} \texttt{List} & : & \star/\mathsf{T} \to \star \\ \texttt{Nil} & : & (a{:}\star/\mathsf{T}).\,\texttt{List}\ a \\ \texttt{Cons} & : & (a{:}\star/\mathsf{T}).\,a \to \texttt{List}\ a \to \texttt{List}\ a \end{array}$$

What about GADTs? Here's an example in Haskell:

```
data Rep a where
  Rint  :: Rep Int
  Rlist :: Rep a → Rep (List a)
```

Although a Haskell programmer writes the data constructors of a GADT with non-parametric result types, in the internal type system it is simpler in the formalization for the result type of a data constructor to take the form $(T\ a_1 \ldots a_n)$, where the $\overline{a}$ are the type parameters, using equality constraints to express the indexing, thus:

$$\begin{array}{ll} \texttt{Rep} & : \star/\mathsf{C} \to \star \\ \texttt{Rint} & : (a{:}\star/\mathsf{C}).(a \sim \texttt{Int}) \Rightarrow \texttt{Rep}\ a \\ \texttt{Rlist} & : (a{:}\star/\mathsf{C}).\forall\,(b : \star/\mathsf{C}).(a \sim \texttt{List}\ b) \Rightarrow \texttt{Rep}\,b \to \texttt{Rep}\ a \end{array}$$

Notice here that `Rep`'s kind expresses that its argument is an *index* (role C) rather than a *parameter* (role T). In fact, the C role for variable a falls out naturally because a appears as argument to the $(\sim_\star)$ constructor (in the type of `Rint` and `Rlist`), which in turn requires it to have role C.

More generally, we use the notation of *telescopes* [de Bruijn 1991] to conveniently express the types of data constructors. Figure 1 defines a telescope Θ like this:

$$\Theta ::= \cdot \mid a{:}\kappa, \Theta \mid \varphi_1 \sim \varphi_2, \Theta \mid \sigma, \Theta$$

A telescope is like a mini-context: a list of type variable bindings, equality propositions (between codes only) and types that classify each argument of the data constructor. (Note that a type variable context Δ is also a telescope.) We define the syntactic sugar $\forall\,\Theta.\sigma$ as follows:

DEFINITION 9 (Telescope syntactic sugar).

$$\begin{array}{lcl} \forall\,\cdot.\sigma & = & \sigma \\ \forall\,(a{:}\kappa, \Theta).\sigma & = & \forall a{:}\kappa.(\forall\,\Theta.\sigma) \\ \forall\,(\varphi_1 \sim \varphi_2, \Theta).\sigma & = & (\varphi_1 \sim \varphi_2) \Rightarrow (\forall\,\Theta.\sigma) \\ \forall\,(\sigma', \Theta).\sigma & = & \sigma' \to (\forall\,\Theta.\sigma) \end{array}$$

At each use of a data constructor (rule EDATACON), we check that the declaration of a data constructor is of the form

$$K{:}\forall\,\overline{a}{:}\overline{\kappa}.(\forall\,\Theta.\,T\,\overline{a})$$

where $\overline{a}{:}\overline{\kappa}$ lists the type parameters of the datatype, Θ describes the types of the arguments to the data constructor, and $T\,\overline{a}$ is syntactic sugar for the application of T to those parameters (i.e. $((T\ a_1)\ldots a_n)$). For example, the `List` type has a single parameter $a{:}\star/\mathsf{T}$, the constructor `Cons` has telescope $(a, \texttt{List}\ a, \cdot)$, and constructs type `List` a. Likewise, the `Rep` type has the parameter $a{:}\star/\mathsf{C}$, the telescope for `Rlist` is $(b{:}\star/\mathsf{C}, a \sim \texttt{List}\ b, (\texttt{Rep}\ b), \cdot)$, and the result type is `Rep` a.

$$\frac{\gamma_0 = \mathbf{sym}\,(\mathbf{nth}\,0\,\gamma) \quad \gamma_1 = \mathbf{nth}\,1\,\gamma}{((\lambda x{:}\sigma_1.e_1) \triangleright \gamma)\; e_2 \rightsquigarrow (\lambda x{:}\sigma_1.(e_1 \triangleright \gamma_1))\;(e_2 \triangleright \gamma_0)} \quad \text{SSPush}$$

$$\frac{\Sigma \vdash \gamma : \forall a{:}\kappa.\sigma_1 \sim \forall a{:}\kappa.\sigma_2 \in \star/\mathsf{T}}{((\Lambda a{:}\kappa.e) \triangleright \gamma)\;\varphi \rightsquigarrow (\Lambda a{:}\kappa.(e \triangleright \gamma@a))\;\varphi} \quad \text{SSTPush}$$

$$\frac{\begin{array}{c}\Sigma \vdash \gamma : (\sim_\eta)\;\varphi_1\;\varphi_2\;\sigma \sim (\sim_\eta)\;\varphi_1'\;\varphi_2'\;\sigma' \in \star/\mathsf{T} \\ \gamma_0 = \mathbf{nth}\,0\,\gamma \quad \gamma_1 = \mathbf{sym}\,(\mathbf{nth}\,1\,\gamma) \quad \gamma_2 = \mathbf{nth}\,2\,\gamma\end{array}}{\begin{array}{l}((\Lambda c{:}\varphi_1 \sim \varphi_2.e) \triangleright \gamma)\;\gamma' \rightsquigarrow \\ \quad (\Lambda c{:}\varphi_1 \sim \varphi_2.(e \triangleright \gamma_2))\;(\gamma_0\,;\,\gamma'\,;\,\gamma_1)\end{array}} \quad \text{SSCPush}$$

$$\frac{\begin{array}{c}\Sigma \vdash \gamma : T\,\overline{\varphi} \sim T\,\overline{\varphi}' \in \star/\mathsf{T} \\ K{:}\forall\,\overline{a}{:}\overline{\kappa}.\sigma\;\in\;\Sigma\end{array}}{\begin{array}{l}\mathbf{case}'_\sigma\;(K\,\overline{\varphi}\,\overline{\rho}) \triangleright \gamma\;\mathbf{of}\;brs \rightsquigarrow \\ \quad \mathbf{case}'_\sigma\;K\,\overline{\varphi}'\;(\overline{\rho} \triangleright \sigma[\overline{a} \mapsto nth\;\gamma])\;\mathbf{of}\;brs\end{array}} \quad \text{SSKPush}$$

Figure 5. Operational Semantics (Push rules)

Telescope notation is also used for case expressions in rule ECase (Figure 2). The type of the scrutinee of the case must be headed by a datatype constant T. Furthermore, for each data constructor that could create a T, there must be a corresponding branch. The (elided) function $\mathrm{Constr}_\Gamma(T)$ looks up the constructors of T from the context. After substituting for the parameters, the branch for data constructor K_i must abstract the same arguments as K_i and return the same result type as the entire case. To make sure that typechecking is syntax directed even when there are no branches (for empty datatypes), the case expression is annotated with its result type σ, and we must check that this type is well-formed in the current context.

3.6 Operational semantics: pushing coercions

The operational semantics of FC_2 is based on a small-step call-by-name operational semantics for a polymorphic lambda calculus. Although this language includes explicit type abstraction and application (as well as explicit coercion proof abstraction and application), types and coercions are not relevant at run time. All such abstractions and applications, as well as uses of coercions $e \triangleright \gamma$ may be erased prior to execution and so impose no run-time costs.

As alluded to above, this operational semantics preserves coercion proofs, which allows us to establish type safety using standard syntactic proofs of progress and preservation (described in the next section). The most important rules of the operational semantics are those that "push" coercions when they appear in active positions so that they do not interfere with reduction. Figure 5 shows the relevant pushing rules. (The complete rules of the operational semantics are listed in the extended version.) In some of the push rules, the coercion must be checked to constrain the types that it proves equal. This checking happens in a global context Σ that contains only the definitions of data constructors, data types, and coercion axioms. Note that these checks are not actually necessary at runtime, in a consistent context these checks will always succeed.

The first three rules show how in an application of a coerced abstraction, the term steps to a new application, where the coercion has been decomposed into a coercion for the body of the abstraction, and a coercion of the argument. For example, consider SSPush. Here, the γ is a coercion between some function types $\sigma_1 \to \sigma_1'$ and $\sigma_2 \to \sigma_2'$. The rule uses $\mathbf{nth}$ and $\mathbf{sym}$ to decompose γ into two coercions, one from $\sigma_2 \sim \sigma_1$ (the order is reversed to account for contra-variance) and one from $\sigma_1 \sim \sigma_2'$. These new coercions can be pushed to the body of the lambda and the function argument, exposing the reduction. Rules SSTPush and SSCPush work analogously.

$\boxed{\Gamma \vdash \overline{\rho} : \Theta}$

$$\frac{\vdash \Gamma}{\Gamma \vdash \cdot : \cdot} \quad \text{TNil}$$

$$\frac{\Gamma \vdash e : \sigma \quad \Gamma \vdash \overline{\rho} : \Theta}{\Gamma \vdash e, \overline{\rho} : (\sigma, \Theta)} \quad \text{TConsE}$$

$$\frac{\Gamma \vdash \varphi : \kappa \quad \Gamma \vdash \overline{\rho} : \Theta[a \mapsto \varphi]}{\Gamma \vdash \varphi, \overline{\rho} : (a{:}\kappa, \Theta)} \quad \text{TConsT}$$

$$\frac{\Gamma \vdash \gamma : \varphi_1 \sim \varphi_2 \in \eta/\mathsf{C} \quad \Gamma \vdash \overline{\rho} : \Theta}{\Gamma \vdash \gamma, \overline{\rho} : (\varphi_1 \sim \varphi_2, \Theta)} \quad \text{TConsC}$$

Figure 6. Telescope rules

Note that SSCPush justifies the kind of $(\sim_\eta)$, which requires that the first two arguments be codes. If we had assigned $(\sim_\eta)$ the parametric kind $\eta/\mathsf{T} \to \eta/\mathsf{T} \to \star/\mathsf{T} \to \star$, then the coercions γ_0 and γ_1 in the rule would both be type coercions. However, type coercions cannot be composed with γ' to form a code coercion, which is the role required for the right hand side of the rule to typecheck.

The last rule, SSKPush, pushes the coercion of a data constructor in the scrutinee position of a case expression into coercions of the arguments of the data constructor. The arguments of a data constructor, notated ρ, can either be an expression, a type, or a coercion.

$$\rho ::= e \mid \varphi \mid \gamma$$

If the declared type of the data constructor K is $\forall\,\overline{a} : \overline{\kappa}.\forall\,\Theta.\,T\,\overline{a}$, and the type parameters are $\overline{\varphi}$, then we know that the arguments can be typed using $\Gamma \vdash \overline{\rho} : \Theta[\overline{a} \mapsto \overline{\varphi}]$ (see Figure 6). However, pushing the coercion changes the type parameters to be $\overline{\varphi}'$, so the new arguments must have type $\Theta[\overline{a} \mapsto \overline{\varphi}']$.

These new arguments are produced by coercing the list of arguments $\overline{\rho}$ with the coercion generated by *lifting* as described above. (The notation $\sigma[\overline{a} \mapsto nth\;\gamma]$ means that variable a_i is lifted to coercion $\mathbf{nth}\;i\;\gamma$.) Once this coercion has been defined by lifting, we use it to coerce the list of arguments of the data constructor with the following operation.

Definition 10. *Define* argument coercion $\overline{\rho} \triangleright \gamma$ *by induction on* $\overline{\rho}$*:*

$$\begin{array}{lcl}\cdot \triangleright \gamma & = & \cdot \\ (e, \overline{\rho}) \triangleright \gamma_1 \to \gamma_2 & = & (e \triangleright \gamma_1), \overline{\rho} \triangleright \gamma_2 \\ (\varphi, \overline{\rho}) \triangleright \forall a{:}\kappa.\gamma_2 & = & \varphi, \overline{\rho} \triangleright \gamma_2 \\ (\gamma, \overline{\rho}) \triangleright (\gamma_1 \sim \gamma_2) \Rightarrow \gamma_3 & = & (\mathbf{sym}\,\gamma_1\,;\,\gamma\,;\,\gamma_2), \overline{\rho} \triangleright \gamma_3\end{array}$$

Argument coercion coerces a list of arguments as described by the following lemma.

Lemma 11. *If* $\Gamma \vdash \overline{\rho} : \Theta$ *and* $\Gamma \vdash \gamma : \forall\,\Theta.\sigma \sim \forall\,\Theta'.\sigma' \in \star/\mathsf{T}$ *then* $\Gamma \vdash (\overline{\rho} \triangleright \gamma) : \Theta'$

4. Type safety and consistency

The FC_2 language supports a straightforward proof of type safety based on the usual preservation and progress theorems. Importantly, the progress theorem holds only for *consistent* contexts—those that cannot equate `Int` and `Char`, for example. Below, we state the progress and preservation theorems and give a precise definition of consistency. In the next subsection, we formulate sufficient conditions for proving that a context is consistent.

4.1 Preservation and progress

The preservation proof for FC_2 is standard, relying on the usual regularity and substitution lemmas for the various judgement forms. For space reasons, we omit those definitions here and instead refer the reader to the extended version.

THEOREM 12 (Preservation). *If* $\Gamma \vdash e_1 : \sigma$ *and* $e_1 \rightsquigarrow e_2$ *then* $\Gamma \vdash e_2 : \sigma$.

The progress theorem holds only for *closed*, *consistent* contexts. A context is *closed* if it does not contain any term variable bindings. We use the metavariable Σ for closed contexts.

The definition of consistency and the canonical forms lemma (necessary to show the progress theorem) are both stated using the notions of uncoerced *values* and their types, *value types*. Value types include all types except those that are headed by a variable a or type function/newtype F. Formally, we define values v and value types τ, with the following grammars:

$$\begin{array}{lcl} \tau & ::= & T \mid (\rightarrow) \mid (\sim_\eta) \mid \forall a{:}\kappa.\varphi \mid \tau\,\varphi \\ v & ::= & \lambda x{:}\sigma.e \mid \Lambda a{:}\kappa.e \mid \Lambda c{:}\varphi_1 \sim \varphi_2.e \mid K\,\overline{\varphi}\,\overline{\rho} \end{array}$$

The canonical forms lemma tells us that the shape of a value is determined by its type:[3]

LEMMA 13 (Canonical Forms). *Say* $\Sigma \vdash v : \sigma$. *Then* σ *is a value type. Furthermore,*

1. *If* $\sigma = \sigma_1 \rightarrow \sigma_2$ *then* v *is* $\lambda x{:}\sigma_1.e$ *or* $K\,\overline{\varphi}\,\overline{\rho}$.
2. *If* $\sigma = \forall a{:}\kappa.\sigma'$ *then* v *is* $\Lambda a{:}\kappa.e$ *or* $K\,\overline{\varphi}\,\overline{\rho}$.
3. *If* $\sigma = (\varphi_1 \sim \varphi_2) \Rightarrow \sigma'$ *then* v *is* $\Lambda c{:}\varphi_1 \sim \varphi_2.e$ *or* $K\,\overline{\varphi}\,\overline{\rho}$.
4. *If* $\sigma = T\,\overline{\varphi}_1$ *then* v *is* $K\,\overline{\varphi}\,\overline{\rho}$.

In FC_2, not all irreducible forms are values. Evaluation can also produce a *coerced value* of the form $v \triangleright \gamma$, which erases to a value when coercions are dropped. To prove the progress theorem, we must reason about what sort of coercion γ could be so that we can appropriately apply the "push" rules in Figure 5. In the statement of the progress theorem, because γ coerces the value v, we know (by ECOERCE and canonical forms) that $\gamma : \tau \sim \sigma$—the left type is a value type τ. *Consistency* of the axiom set assures us that if σ is also a value type, it must have the same head form.

DEFINITION 14 (Consistency). *A context* Γ *is consistent if whenever* $\Gamma \vdash \gamma : \tau_1 \sim \tau_2 \in \eta/\mathsf{T}$ *it is the case that*

1. *If* τ_1 *is* $T\,\overline{\varphi}_1$ *then* τ_2 *is* $T\,\overline{\varphi}_2$.
2. *If* τ_1 *is* $(\rightarrow)\,\overline{\varphi}_1$ *then* τ_2 *is* $(\rightarrow)\,\overline{\varphi}_2$.
3. *If* τ_1 *is* $(\sim_\eta)\,\overline{\varphi}_1$ *then* τ_2 *is* $(\sim_\eta)\,\overline{\varphi}_2$.
4. *If* τ_1 *is* $\forall a{:}\kappa.\sigma_1$ *then* τ_2 *is* $\forall a{:}\kappa.\sigma_2$.

Putting these observations together, we obtain:

THEOREM 15 (Progress). *If* Σ *is consistent and* $\Sigma \vdash e_1 : \sigma$ *and* e_1 *is not a value* v *or a coerced value* $v \triangleright \gamma$, *then there exists an* e_2 *such that* $e_1 \rightsquigarrow e_2$.

4.2 Determining consistency

Although the previous subsection gives a definition of when contexts are consistent, it does not provide any mechanism for determining whether a set of axioms leads to a consistent context.

This subsection defines *sufficient* conditions (written $\mathbf{Good}\,\Gamma$) for establishing context consistency—these conditions are not the only way to show consistency (they are not *necessary*) but they are permissive enough to cover the axioms generated by compilation of type family declarations and newtype definitions.

[3] Note that all forms of value type must include *partial* applications of data constructors.

$\boxed{\Gamma \vdash \varphi \rightsquigarrow \varphi' \in \kappa}$

$$\frac{\Gamma \vdash a : \overline{\kappa} \rightarrow \eta/R \quad \Gamma \vdash \overline{\varphi} \rightsquigarrow \overline{\varphi}' \in \overline{\kappa}/R}{\Gamma \vdash a\,\overline{\varphi} \rightsquigarrow a\,\overline{\varphi}' \in \eta/R}\ \text{RVAR}$$

$$\frac{H{:}\overline{\kappa} \rightarrow \eta \in \Gamma \quad \Gamma \vdash \overline{\varphi} \rightsquigarrow \overline{\varphi}' \in \overline{\kappa}/R}{\Gamma \vdash H\,\overline{\varphi} \rightsquigarrow H\,\overline{\varphi}' \in \eta/R}\ \text{RCONST}$$

$$\frac{\Gamma, a{:}\kappa \vdash \sigma \rightsquigarrow \sigma' \in \star/R}{\Gamma \vdash \forall a{:}\kappa.\sigma \rightsquigarrow \forall a{:}\kappa.\sigma' \in \star/R}\ \text{RALL}$$

$$\frac{\begin{array}{l} H{:}\overline{\kappa}_1 \rightarrow (\overline{\kappa}_2 \rightarrow \eta) \in \Gamma \\ \Gamma \vdash \overline{\varphi}_1 \rightsquigarrow \overline{\varphi}'_1 \in \overline{\kappa}_1/R \\ \Gamma \vdash \overline{\varphi}_2 \rightsquigarrow \overline{\varphi}'_2 \in \overline{\kappa}_2/R \\ \Gamma \vdash c\,\overline{\psi} : H\,\overline{\varphi}'_1 \sim \sigma \in \overline{\kappa}_2 \rightarrow \eta/R \end{array}}{\Gamma \vdash H\,\overline{\varphi}_1\,\overline{\varphi}_2 \rightsquigarrow \sigma\,\overline{\varphi}'_2 \in \eta/R}\ \text{RRED}$$

$\boxed{\Gamma \vdash \overline{\varphi} \rightsquigarrow \overline{\varphi}' \in \overline{\kappa}/R}$

$$\frac{}{\Gamma \vdash \cdot \rightsquigarrow \cdot \in \cdot/R}\ \text{RNIL}$$

$$\frac{\Gamma \vdash \varphi \rightsquigarrow \varphi' \in \eta/min(R_1, R_2) \quad \Gamma \vdash \overline{\varphi} \rightsquigarrow \overline{\varphi}' \in \overline{\kappa}/R_1}{\Gamma \vdash \varphi, \overline{\varphi} \rightsquigarrow \varphi', \overline{\varphi}' \in \eta/R_2, \overline{\kappa}/R_1}\ \text{RCONS}$$

Figure 7. Type rewriting

As in the previous version of FC, we show consistency by (i) defining a rewrite system for types and (ii) proving that two types are joinable (share a common reduct) if and only if there is some coercion proof between those types. The rewrite system guarantees that value types preserve their head form throughout rewriting and therefore value types with different head forms can never be equated.

4.2.1 Rewriting and joinability

Figure 7 gives our rewrite relation, which is a variant of parallel reduction—it looks throughout the type, trying to fire reductions. The reduction of a type headed by a constant H is implemented in rule RRED. If, after the arguments to H have been reduced to $\overline{\varphi}'$, there is some instantiation of an axiom such that H applied to some prefix of the $\overline{\varphi}'$ matches the left-hand side of the coercion, then the type may reduce to the right-hand-side type (RRED). Importantly, rewriting (very much like coercibility) takes roles into account: rewriting occurs at some role R, which specifies what axioms are available. For example, at role T we can rewrite a newtype `Age` to its definition `Int`, but at role C, we cannot.

Notice that rules RCONST and RRED overlap. The term $H\,\overline{\varphi}$ may reduce using a matching axiom for H (via rule RRED), but not necessarily (via RCONST). With rule RCONST, the rewrite relation is reflexive.

LEMMA 16. *If* $\Gamma \vdash \varphi : \kappa$ *then* $\Gamma \vdash \varphi \rightsquigarrow \varphi \in \kappa$.

The *joinability* relation, below, is simply reduction to a common reduct after zero or more steps.

DEFINITION 17 (Joinable). *Two types are* joinable *if they share a common reduct. Define* $\Gamma \vdash \varphi_1 \Leftrightarrow \varphi_2 \in \kappa$ *if* $\Gamma \vdash \varphi_1 \rightsquigarrow^* \varphi \in \kappa$ *and* $\Gamma \vdash \varphi_2 \rightsquigarrow^* \varphi \in \kappa$.

Reflexivity of the rewrite relation is crucial for joinability of type applications with multiple arguments. Consider the axiom

$$\texttt{c} : \texttt{F Int} \sim \texttt{(F Int, F Int)}/\mathsf{C}$$

and the following two types:

$$\varphi_1 = \texttt{T (F Int) (F Int, F Int)}$$
$$\varphi_2 = \texttt{T (F Int, F Int) (F Int)}$$

We'd certainly like φ_1 and φ_2 to be reducible to a common reduct (after all, there exists a coercion term between them), but the first argument of T in φ_1 "lags behind" compared to the first argument of T in φ_2, whereas the second argument in φ_1 "advances ahead" compared to the second argument in φ_2. Reflexivity allows us to freeze the reduction of the second argument in φ_1 while reducing the first, and similarly for φ_2—which allows us to join the two types.

4.2.2 Soundness and completeness of the rewrite relation

We now give sufficient conditions on contexts which ensure that the rewrite relation of Figure 7 is sound and complete with respect to coercibility. Hence, these are sufficient conditions for consistency.

DEFINITION 18 (Good contexts). *We have* $\mathbf{Good}\,\Gamma$ *when the following conditions hold:*

1. *All axioms in* Γ *are of the form* $c : \Delta.\, F\,\overline{\varphi} \sim \psi/R$. *Let* $\overline{a} = dom(\Delta)$ *and* $\overline{\kappa}$ *be the corresponding kinds. For every well kinded* $\overline{\psi}$ *and result types* $\overline{\varphi}'$, *such that* $\Gamma \vdash (\overline{\varphi}[\overline{a} \mapsto \overline{\psi}]) \rightsquigarrow \overline{\varphi}' \in \overline{\kappa}'/R$ *it must be* $\overline{\varphi}' = \overline{\varphi}[\overline{a} \mapsto \overline{\psi}']$ *for some* $\overline{\psi}'$ *with* $\Gamma \vdash \overline{\psi} \rightsquigarrow \overline{\psi}' \in \overline{\kappa}/R$.
2. *There is no overlap between axioms. For each* $F\,\overline{\varphi}$ *there exists at most one prefix* $\overline{\varphi}_1$ *of* $\overline{\varphi}$ *such that there exist* c, ψ *and* $\overline{\psi}$ *where* $\Gamma \vdash c\,\overline{\psi} : F\,\overline{\varphi}_1 \sim \psi \in \kappa$. *This* c *is unique for every matching* $F\,\overline{\varphi}_1$.

The first condition above restricts the declared arguments of type functions to behave like patterns. It merely states that all the reductions that can possibly happen when we substitute types for Δ do not involve reductions from $\overline{\varphi}$ but only reductions *inside* those substituted types. This condition is a generalized (and more lenient) form of the current GHC restrictions on type family declarations, which only allow value types or variables as $\overline{\varphi}$. The second condition is simply a strong non-overlapping condition.

Interestingly, the first condition on the arguments to type functions restricts the kind that a type function may have. For example, recall the axioms for F from Section 2.2:

$$\texttt{axF1} : \texttt{F Int} \sim \texttt{Bool}/\mathsf{C}$$
$$\texttt{axF2} : \texttt{F Age} \sim \texttt{Char}/\mathsf{C}$$

For this to be a $\mathbf{Good}$ axiom set, the kind of F must be $\star/\mathsf{C} \to \star$ because the newtype Age is irreducible only in role C.
In the rest of this section, we sketch the proof that our conditions are sufficient for consistency. Soundness is a straightforward proof.

THEOREM 19 (Soundness). *If if* $\mathbf{Good}\,\Gamma$ *and* $\Gamma \vdash \varphi_1 \rightsquigarrow \varphi_2 \in \kappa$ *then there is some* γ *such that* $\Gamma \vdash \gamma : \varphi_1 \sim \varphi_2 \in \kappa$.

The three key lemmas of the completeness proof are that joinability is preserved under application and substitution and a local diamond property of rewriting.

LEMMA 20 (Application). *If* $\mathbf{Good}\,\Gamma$ *and* $\Gamma \vdash \varphi_1 \Leftrightarrow \varphi_1' \in (\eta_1/R_1 \to \eta_2)/R_2$ *and* $\Gamma \vdash \varphi_2 \Leftrightarrow \varphi_2' \in \eta_1/min(R_1, R_2)$ *then* $\Gamma \vdash \varphi_1\,\varphi_2 \Leftrightarrow \varphi_1'\,\varphi_2' \in \eta_2/R_2$.

THEOREM 21 (Local diamond property). *If* $\mathbf{Good}\,\Gamma$ *and* $\Gamma \vdash \varphi \rightsquigarrow \varphi_1 \in \kappa$ *and* $\Gamma \vdash \varphi \rightsquigarrow \varphi_2 \in \kappa$ *then there exists a* φ_3 *such that* $\Gamma \vdash \varphi_1 \rightsquigarrow \varphi_3 \in \kappa$ *and* $\Gamma \vdash \varphi_2 \rightsquigarrow \varphi_3 \in \kappa$.

LEMMA 22 (Substitution). *If* $\mathbf{Good}\,\Gamma$ *and* $\Gamma, a{:}\kappa, \Delta \vdash \sigma \rightsquigarrow^* \sigma' \in \kappa'$ *and* $\Gamma \vdash \varphi \rightsquigarrow^* \varphi' \in \kappa$, *then there is some* $\Gamma, \Delta \vdash \sigma[a \mapsto \varphi] \Leftrightarrow \sigma'[a \mapsto \varphi'] \in \kappa'$.

From these lemmas we see that joinability is complete.

THEOREM 23 (Completeness). *If* $\mathbf{Good}\,\Gamma$ *and* $\Gamma \vdash \gamma : \varphi_1 \sim \varphi_2 \in \kappa$ *then* $\Gamma \vdash \varphi_1 \Leftrightarrow \varphi_2 \in \kappa$.

THEOREM 24 (Consistency). *If* $\mathbf{Good}\,\Gamma$ *then* Γ *is consistent.*

Proof Suppose $\Gamma \vdash \gamma : \tau_1 \sim \tau_2 \in \star/R$, where τ_1 and τ_2 are two value types. By completeness, we have that those two types are joinable, i.e. that there is some φ such that $\Gamma \vdash \tau_1 \rightsquigarrow^* \varphi \in \star/R$ and $\Gamma \vdash \tau_2 \rightsquigarrow^* \varphi \in \star/R$. However, by inversion on the rewriting relation, we see that it preserves the head forms of value types (since there exist no axioms for those by the first condition of $\mathbf{Good}\,\Gamma$). Thus, τ_1 and τ_2 (and φ) have the same head form.

A different novelty of our approach compared to previous work [Sulzmann et al. 2007] is that establishing the completeness theorem, and therefore type soundness, does not depend on strong normalization. Our definition of $\mathbf{Good}\,\Gamma$ that (i) forces the declared type family arguments to behave like patterns, and (ii) imposes strong (stronger than the previous work) non-overlapping conditions, is sufficient to show completeness.

With this new approach, a programmer can be confident that, even in the presence of possibly non-terminating type functions, if the compiler can show that the program is well-typed then the program will not crash. Although type checking is undecidable in general without strong normalization, there may be many programs that the compiler *can* actually type check, and this approach shows that those programs are type sound.

5. Compilation from source Haskell

In the previous sections we have informally presented the translations of Haskell source features such as datatype, type family, and **newtype** declarations into FC_2. We summarize them here:

$$\begin{array}{ll}
\textbf{data}\ T\ \Delta\ \textbf{where}\ \overline{K_i : \sigma} & \mapsto \\
\quad T : \forall\Delta.\star, \overline{K_i : \Delta.\sigma} & \\
\textbf{type family}\ F : \overline{\eta/\mathsf{C}} \to \eta & \mapsto \\
\quad F : \overline{\eta/\mathsf{C}} \to \eta & \\
\textbf{type instance}\ \Delta.F\ \overline{\varphi} = \psi & \mapsto \\
\quad cF : \Delta.F\ \overline{\varphi} \sim \psi \in \eta/\mathsf{C} & \\
\textbf{newtype}\ \Delta.M\ \overline{a} = \texttt{MkM}\ \varphi & \mapsto \\
\quad M : \Delta.\star, \texttt{MkM} : \Delta.M\ \overline{a} \sim \varphi \in \star/\mathsf{T} &
\end{array}$$

In addition, uses of the data constructor of a **newtype**, both in terms and patterns, are translated to a use of the corresponding coercion in the obvious way. The important parts of these definitions are that (i) type families accept code arguments, (ii) type instances give rise to code equalities, and (iii) newtype definitions give rise to type equality axioms. The bindings generated in this way can be easily checked for well-formedness. If, in addition, the resulting context is $\mathbf{Good}$ (see Section 4.2)—the only possible problem is the potential to generate overlapping **type instance** declarations—then the context is consistent, which in turn guarantees type safety.

Notice that the source language features in this translation have been *already annotated* with their kinds. This is a reasonable assumption. Prior to type inference, which translates a source declaration to an FC_2 binding, we must have determined the kinds involved in the declarations. For the purposes of this paper, we assume that the kinds are given—in practice they would be the output of a kind inference process, potentially guided by the user to disambiguate role information and higher-order kinds.

Newtype deriving Generative type abstraction is achieved in Haskell using the **newtype deriving** mechanism, which allows type classes to be automatically lifted to new types. For instance, we may write

```
newtype Age = MkAge Int deriving Eq
```

The type class `Eq a` is a standard class in Haskell (signature, in ML terminology) that defines one method, `eq :: a → a → Bool`. For the type checker, a type class is nothing but a record type containing the method `eq`. The **deriving** line automatically generates an implementation of `Eq Age` from a pre-existing instance `Eq Int`. This can be done by simply applying a coercion `Eq Int` ∼ `Eq Age` to the old record. It is straightforward to construct this coercion: lift the `Age` ∼ `Int` axiom from the **newtype** definition over the `Eq` constructor and apply symmetry. Importantly, this lifting is safe because `Eq` has a *parametric kind* of the form $\star/\mathsf{T} \to \star$. However, if a type class has a *indexed kind* of the form $\star/\mathsf{C} \to \star$, **newtype deriving** is no longer sound. For example

```
class C x where
  op :: F x → x
instance C Int where ...
newtype Age = MkAge Int deriving C
```

It is unsound to lift the `C Int` implementation to a `C Age`, precisely because `x` has role C. The documented GHC bug mentioned earlier arises precisely as the result of such an unsound lifting over a non-parametric type class.

6. Discussion

6.1 Relaxing decomposition

Recall the coercion decomposition rule, CNTHT, from Figure 4. This rule allows us to deconstruct an equality of the form $\Gamma \vdash \gamma : T\,\overline{\varphi} \sim T\,\overline{\psi} \in \eta/\mathsf{T}$. In effect, it asserts that data constructors are injective. The rule is important because it is used in the operational semantics to ensure subject reduction. However, the decomposition rule may be somewhat restrictive for some Haskell source programs. Consider the following:

```
newtype M a = MkM (Maybe [a])
data Eq a b where EQ :: Eq a a
f :: Eq (M a) (M b) → a → b
f EQ x = x
```

Pattern matching against the `EQ` constructor introduces a coercion between `M a` and `M b`, which cannot be decomposed using the CNTHT rule to obtain `a` ∼ `b`, so this program cannot be typed. Nevertheless, *we know* that `M` is injective, because `M a` is defined to be equal to `Maybe [a]`, which is clearly injective.

On the other hand, the following **newtype** is *not* injective.

```
type instance G a  = Char
newtype N a = MkN (G a)
```

Here, it is possible to derive $\Gamma \vdash \gamma$: `N Int` ∼ `N Char` $\in \star/\mathsf{T}$, using the axiom for `G`, even though `Int` is not coercible to `Char`.

It turns out that **newtype** definitions are *always* injective with respect to code equality, but they might not be injective with respect to type equality (as illustrated by the two examples above). Thus it would be sound and potentially useful (but not necessary for type soundness) to introduce yet another decomposition rule for **newtype** definitions that takes advantage of injectivity with respect to codes (we use metavariable N newtypes):

$$\frac{\begin{array}{l}\Gamma \vdash \gamma : N\,\overline{\varphi} \sim N\,\overline{\psi} \in \eta/\mathsf{C} \\ N{:}\overline{\kappa} \to \eta \in \Gamma \\ \eta'/R' = nth\ k\ \overline{\kappa}\end{array}}{\Gamma \vdash \mathbf{nth}\,k\,\gamma : nth\ k\ \overline{\varphi} \sim nth\ k\ \overline{\psi} \in \eta'/R}\ \text{CNTHN}$$

The only subtle part of this rule is that R is not related to R', since the equality $\Gamma \vdash \gamma : N\,\overline{\varphi} \sim N\,\overline{\psi} \in \eta/\mathsf{C}$ is a C-equality (and $min(\mathsf{C}, R') = \mathsf{C}$).

Arguably, decomposition for injective type functions is also desirable, were we able to effectively specify and check that property.

6.2 Other technical differences of FC_2 from System FC

The intermediate language FC_2 described in this paper is a significant modification of System FC [Sulzmann et al. 2007] due to the introduction of codes. However, FC_2 also makes a number of *technical simplifications*:

- The original System FC presentation includes *coercion kinds*, $\sigma_1 \sim \sigma_2$. The original coercion language includes three additional constructs, one to coerce coercions, and two more to decompose coercion kinds. By treating $(\varphi_1 \sim \varphi_2) \Rightarrow \varphi_3$ as the application of a constructor $(\sim_\eta)$ we no longer need any of these constructs in the operational semantics, nor have we identified any uses of these constructs that are not encodable.
- The operational semantics rules of FC_2 in Figure 5 not only use simpler coercion constructs, but are also expressed without need for substitutions, contrary to their original FC versions.
- FC_2 replaces the FC coercions **left** and **right**, which decomposed arbitrary type applications, by **nth**, which decomposes only the application of a datatype constructor. This allows us to lift a tiresome restriction in in System FC, namely that type functions were required to be saturated. Type family saturation was necessary in FC, in order to prevent the decomposition of equalities as `F` $a \sim$ `Maybe` $[a]$ via **left** or **right**. Allowing unsaturated functions increases the expressiveness of FC_2, because we can now abstract over type functoins, and also opens new directions for future research on type inference in the presence of unsaturated type functions.
- However, using **nth** instead of **left** and **right** does carry a small price. Generalizing the example from Section 6.1, should this program be well typed?

  ```
  data Eq a b where EQ :: Eq a a
  f :: Eq (p q) (r s) → q → s
  f EQ x = x
  ```

  To type it we must decompose a proof that $p\ q \sim r\ s$ to get a proof that $q \sim s$, which **right** could do, but **nth** cannot.

Some other differences are *presentational*:

- System FC used a common syntax for types and coercions, which is a convenient pun, but has turned out to be more confusing than helpful. In FC_2 we use a distinct syntax for types and coercions (Figure 1).
- In FC_2 we define top-level axiom schemes $c : \Delta.\,\varphi_1 \sim \varphi_2/R$ directly, and fully instantiate them at every occurrence with the form $c\ \overline{\gamma}$ (Figure 1). System FC instead defined a top-level axiom scheme as an equality between polytypes, thus $c : \forall\Delta.\varphi_1 \sim \forall\Delta.\varphi_2$. Here again FC is confusing (but not wrong) so in FC_2 we opt for telling the story more directly, albeit with slightly more syntax. Moreover the kinding rules for $\forall$ (PALL and CALL) can insist that the body of the forall has kind $\star$ as is conventional.

- Using telescopes in the treatment of datatypes simplifies the operational semantics rules but is also (only slightly) more expressive: The types of data constructors do not have to have their quantified variables preceding their coercion and term arguments. Instead, telescopes allow arbitrary interleavings.

7. Related work

Previous work on System FC [Sulzmann et al. 2007] discusses a significant amount of related work, in typed languages with explicit proof witnesses [Licata and Harper 2005; Shao et al. 2005], or in calculi that support coercions [Breazu-Tannen et al. 1991]. Below, we present related work in generativity and abstraction, type-indexed constructs and the separation between codes and types.

Generativity, abstraction, and module systems Generativity and abstraction has been studied extensively in the context of ML module systems [Milner et al. 1997]. Russo shows how generativity in module systems is connected to existential quantification [Russo 1999] and Dreyer [2005] has studied this connection in the presence of recursive modules. Montagu and Rémy [2009] refine this connection by introducing "open" existential types. Rossberg [2008] uses flexible generativity to explain ML-style module sealing.

Type abstraction can be understood in terms of dynamic name generation [Rossberg 2003; Vytiniotis et al. 2005], which can re-establish abstraction properties in languages with dynamic type analysis. Neis et al. [2009] prove a parametricity theorem in this setting. In addition, they use a translation from polymorphism to generative types to establish the parametric behavior of certain functions although they work in a non-parametric language.

Although many of these languages support type generativity and non-parametric features, they do not exhibit the soundness problems described in the paper, mainly due to the absence of type-level type dispatching. Nevertheless, the techniques developed in the aforementioned related work would be valuable in the formal study of the parametricity properties of FC_2.

Type-indexed types Although many systems for generic programming support dynamic computation based on types, very few systems allow the structure of *types* to be destructed to produce other types. However, such facility is often necessary to describe the *type* of generic programs. For example, Harper and Morrisett include a `Typerec` operator to their typed intermediate language λ_i^{ML} [Harper and Morrisett 1995], to describe type-directed optimizations. (They credit NuPRL's mechanism of "Universe Elimination" in NuPRL as the inspiration for this operation [Constable 1982; Constable and Zlatin 1984].)

To support generic programming in source languages, Hinze, Jeuring and Löh added Type-Indexed Datatypes [Hinze et al. 2002] to the Generic Haskell front end. In later work, Chakravarty et al. [2005b] introduced associated data families in GHC, which are type-indexed datatypes associated with type class instances. Extending this work, they later introduced associated type synonyms [Chakravarty et al. 2005a], which are proper type-level functions with instances associated with type class instances. Currently, the source language of GHC also supports standalone type-level type functions, often referred to as indexed type families [Kiselyov et al. 2010; Schrijvers et al. 2008], a feature that we have extensively used in our presentation.

Codes, types, and interpretations Our distinction between codes and types—and our terminology—is inspired by similar notions in intuitionistic type theory [Benke et al. 2003; Dybjer 2000; Martin-Löf 1975]. There, types (sets) are constructed as the recursive interpretation of codes, which inhabit inductively constructed *code universes*. A **`newtype`** definition can be viewed as giving rise to a new code, inhabiting an open universe of codes, and whose interpretation coincides with the interpretation of its definition.

Languages based on dependent type theory, such as Agda [Bove et al. 2009] or Coq [The Coq Team], naturally offer type-level computation to construct types, but they allow elimination of codes only, not types. Therefore, they do not exhibit the same soundness problem, as the expressiveness of these languages can readily enforce the distinction between types and codes. The disadvantage is the extra programming verbosity of explicit definitions and interpretations of codes. To better support generic programming, the dependently-typed language Epigram [Chapman et al. 2010] identifies types with their code universes.

The LX language [Crary and Weirich 1999] also uses universe constructions to solve problems with type-directed compilation. When the type translation in a compiler pass is not the identity then type dispatch must be compiled to code dispatch (so the generated code can dispatch on source types instead of target types). The interpretation of codes is then the type translation. To support universes, LX includes datakinds (for codes) and primitive recursive functions over datakinds (for their interpretation at types). In LX, source types `Age` and `Int` would be mapped to definable codes `AgeCode` and `IntCode`, and would be accompanied by an interpretation function such that `interp(AgeCode)` equals `Int` and `interp(IntCode)` equals `Int`. Therefore, the problem with generativity would not show up in that context. If one wanted to solve the problem in this paper along the LX lines, one would have to translate source Haskell types to void types that stand in as codes and handle the `interp()` function as any other type function. This function, as well as interpreting the codes as types, would have to be accompanied with suitable congruence axioms, like `interp(T t)` $\sim$ `T (interp(t))`. Explicitly introducing these axioms means that coercions would be significantly more verbose. Our system dispenses with an explicit `interp()` function by conveniently using the roles in the judgements to determine whether we wish to derive an equality between codes or between their interpretations.

8. Conclusions and future work

In this paper we have identified a problem for the safe interaction of flexible type generativity and type-level computation. We have proposed a solution that distinguishes between indexed and parametric type contexts, by extending the language of kinds, and formalized the solution in the FC_2 language. We have several avenues for future research in mind, which we outline below.

Source language technology We would like to work on ways to expose the FC_2 expressive features to programmers. Specific directions are: enriching the kind declarations with the ability to declare parametric or indexed type-level constructs, introducing type family injectivity annotations, extending kind inference with roles, and extending type inference to support unsaturated functions using the more sophisticated kinds.

Enriching the universes of codes with terms We are currently working on enriching the universe of codes with constants or functions drawn from the *term* syntax, such as data constructors, in order to enable direct dependently-typed programming in Haskell.

More roles Lemma 6 asserts that the equivalence classes induced by T-equality are refined by C-equality. However, our approach readily extends to arbitrary lattices of roles with gradually more refined equivalence classes as we move down the $\preceq$ relation.

Consider, for example, a lattice with new roles C_1 and C_2, each lying between C and T, but incomparable with each other. These two roles model partial knowledge of newtype equalities. For example, perhaps C_1 identifies `Age` and `Int` but distinguishes

a different newtype `Moo` from its definition `Bool`. Conversely, C_2 can identify `Moo` and `Bool` and distinguish `Age` and `Int`. These more precise roles could be used to give more precise types to nonparametric functions by identifying exactly which newtype equivalences they do and do not respect. However, we have not yet explored the practical implications of this precision.

Furthermore, one could explore adding a role above T. Type functions with kind $\star/\mathsf{T} \to \star$, are not necessarily parametric in FC_2. For example, because `G` below does not match *newtypes* it may be assigned either kind $\star/\mathsf{T} \to \star$ or $\star/\mathsf{C} \to \star$.

$$\texttt{axG1} : \texttt{G Int} \sim \texttt{Bool}/\mathsf{C}$$
$$\texttt{axG2} : \texttt{G Bool} \sim \texttt{Char}/\mathsf{C}$$

To distinguish truly parametric functions from those like `G`, we could add a role P, induced by the relation that equates all types of the same kind. As this equivalence is coarser than T, it could not (and should not) participate in coercions. However, this new role would provide a yet more descriptive kind for type functions.

Acknowledgements

We thank Brent Yorgey and the attendees of the Type System Wrestling sessions at MSR Cambridge for many useful discussions. We wrote this paper using the Ott tool. This work was partially supported by grants from DARPA (CSSG) and NSF (0910786) and a Sloan Foundation fellowship.

References

M. Benke, P. Dybjer, and P. Jansson. Universes for generic programs and proofs in dependent type theory. *Nordic J. of Computing*, 10(4):265–289, 2003. ISSN 1236-6064.

A. Bove, P. Dybjer, and U. Norell. A brief overview of agda — a functional language with dependent types. In *TPHOLs '09: Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics*, pages 73–78, Berlin, Heidelberg, 2009. Springer-Verlag.

V. Breazu-Tannen, T. Coquand, C. A. Gunter, and A. Scedrov. Inheritance as implicit coercion. *Information and Computation*, 93:172–221, 1991.

M. M. T. Chakravarty, G. Keller, and S. Peyton Jones. Associated type synonyms. In *ICFP '05: Proceedings of the Tenth ACM SIGPLAN International Conference on Functional Programming*, pages 241–253, New York, NY, USA, 2005a. ACM.

M. M. T. Chakravarty, G. Keller, S. Peyton Jones, and S. Marlow. Associated types with class. *SIGPLAN Not.*, 40(1):1–13, 2005b. ISSN 0362-1340.

J. Chapman, P. Évariste Dagand, C. McBride, and P. Morris. The gentle art of levitation. In *Proceedings of the Fifteenth ACM SIGPLAN International Conference on Functional Programming (ICFP '10)*, Baltimore, MD, USA, September 2010. To appear.

J. Cheney and R. Hinze. First-class phantom types. CUCIS TR2003-1901, Cornell University, 2003.

R. L. Constable. Intensional analysis of functions and types. Technical Report CSR-118-82, Department of Computer Science, University of Edinburgh, June 1982.

R. L. Constable and D. R. Zlatin. The type theory of PL/CV3. *ACM Transactions on Programming Languages and Systems*, 6(1):94–117, Jan. 1984.

K. Crary and S. Weirich. Flexible type analysis. In *Proceedings of the Fourth ACM SIGPLAN International Conference on Functional Programming*, pages 233–248, Paris, France, Sept. 1999.

N. G. de Bruijn. Telescopic mappings in typed lambda calculus. *Inf. Comput.*, 91(2):189–204, 1991. ISSN 0890-5401.

D. Dreyer. Recursive type generativity. In *ICFP '05: Proceedings of the Tenth ACM SIGPLAN International Conference on Functional Programming*, pages 41–53, New York, NY, USA, 2005. ACM.

P. Dybjer. A general formulation of simultaneous inductive-recursive definitions in type theory. *Journal of Symbolic Logic*, 65(2):525–549, 2000.

C. V. Hall, K. Hammond, S. L. Peyton Jones, and P. L. Wadler. Type classes in Haskell. *ACM Transactions on Programming Languages and Systems*, 18(2):109–138, Mar. 1996.

R. Harper and G. Morrisett. Compiling polymorphism using intensional type analysis. In *POPL '95: Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 130–141, New York, NY, USA, 1995. ACM. ISBN 0-89791-692-1.

R. Hinze, J. Jeuring, and A. Löh. Type-indexed data types. In B. M. Eerke Boiten, editor, *Proceedings of the Sixth International Conference on Mathematics of Program Construction (MPC 2002)*, pages 148–174, Dagstuhl, Germany, July 2002.

O. Kiselyov, S. Peyton Jones, and C. Shan. *Fun with type functions*. Springer, 2010.

D. R. Licata and R. Harper. A formulation of dependent ML with explicit equality proofs. Technical Report CMU-CS-05-178, Carnegie Mellon University Department of Computer Science, 2005.

P. Martin-Löf. An intuitionistic theory of types: Predicative part. In *Proceedings of the Logic Colloquium, 1973*, volume 80 of *Studies in Logic and the Foundations of Mathematics*, pages 73–118. North-Holland, 1975.

R. Milner, M. Tofte, and D. Macqueen. *The Definition of Standard ML*. MIT Press, Cambridge, MA, USA, 1997. ISBN 0262631814.

B. Montagu and D. Rémy. Modeling abstract types in modules with open existential types. In *POPL '09: Proceedings of the 36th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 354–365, New York, NY, USA, 2009. ACM.

G. Neis, D. Dreyer, and A. Rossberg. Non-parametric parametricity. In *ICFP '09: Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming*, pages 135–148, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-332-7.

S. Peyton Jones et al. The Haskell 98 language and libraries: The revised report. *Journal of Functional Programming*, 13(1):0–255, Jan 2003. `http://www.haskell.org/definition/`.

S. L. Peyton Jones, D. Vytiniotis, S. Weirich, and G. Washburn. Simple unification-based type inference for GADTs. In *International Conference on Functional Programming (ICFP)*, Portland, OR, USA, Sept. 2006.

B. C. Pierce, editor. *Advanced Topics in Types and Programming Languages*. MIT Press, 2005.

A. Rossberg. Dynamic translucency with abstraction kinds and higher-order coercions. *Electr. Notes Theor. Comput. Sci.*, 218:313–336, 2008.

A. Rossberg. Generativity and dynamic opacity for abstract types. In *PPDP '03: Proceedings of the 5th ACM SIGPLAN International Conference on Principles and Practice of Declaritive Programming*, pages 241–252, New York, NY, USA, 2003. ACM. ISBN 1-58113-705-2.

C. V. Russo. Non-dependent types for Standard ML modules. In *PPDP '99: Proceedings of the International Conference PPDP'99 on Principles and Practice of Declarative Programming*, pages 80–97, London, UK, 1999. Springer-Verlag. ISBN 3-540-66540-4.

T. Schrijvers, S. Peyton Jones, M. Chakravarty, and M. Sulzmann. Type checking with open type functions. In *ICFP '08: Proceeding of the 13th ACM SIGPLAN international conference on Functional programming*, pages 51–62, New York, NY, USA, 2008. ACM.

Z. Shao, V. Trifonov, B. Saha, and N. Papaspyrou. A type system for certified binaries. *ACM Trans. Program. Lang. Syst.*, 27(1):1–45, 2005.

M. Sulzmann, M. M. T. Chakravarty, S. Peyton Jones, and K. Donnelly. System F with type equality coercions. In *TLDI '07: Proceedings of the 2007 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation*, pages 53–66, New York, NY, USA, 2007. ACM.

The Coq Team. *Coq*. URL `http://coq.inria.fr`.

D. Vytiniotis, G. Washburn, and S. Weirich. An open and shut typecase. In *ACM SIGPLAN Workshop in Types in Language Design and Implementation*, Long Beach, CA, USA, Jan. 2005.

H. Xi, C. Chen, and G. Chen. Guarded recursive datatype constructors. In *POPL*, pages 224–235, 2003.